Concurrencia

Pau Arlandis Martínez

Sobre las normas

Profesores

- Ángel Herranz – 2309
- Julio Mariño – 2308

Cada lunes se entregará un problema que debe resolverse antes del jueves. Únicamente sirven para practicar y para salvar a alguien que esté casi aprobado. 10 ejercicios.

Parte teórica → 50% (2 exámenes)

Parte práctica → 50% (2 partes) → En grupos de 2 personas.

Cada examen debe superarse con al menos un 4 para hacer media.

Tema 1

Ejercicio 0

Leer *Concepts and Notations for Concurrent Programming* de G.R. Andrews y F.B. Schneider (1983) hasta la sección 3.2 (incluida).

En esta asignatura (y en la informática en general) es bueno conocer los trabajos de tres nombres propios:

- Djsktra
- Hoare
- Knuth

Puesto que crean toda la base teórica y conceptual de la informática.

En Concurrencia debemos tener dos conceptos de Java absolutamente claros:

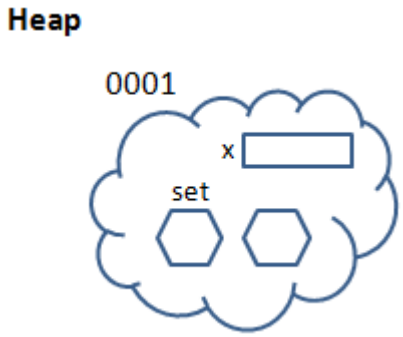
- Scope o ámbito de variables.
- Variable vs Objeto.

Artículo recomendado de la semana sobre neurociencia y aprendizaje tecnológico: *Unskilled and unaware of it*

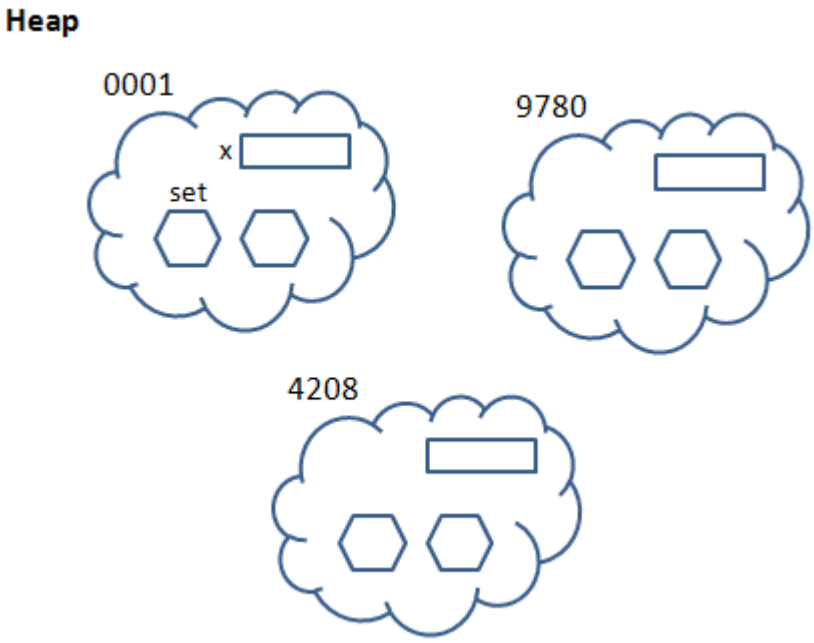
Ejemplo de lectura de lenguaje Java

Tenemos tres variables: a_1, a_2 y a_3. Podemos verlas como cajas que contienen direcciones de memoria. Al iniciar la máquina estas cajas están vacías, tienen una dirección de memoria (en el heap) que es 0.

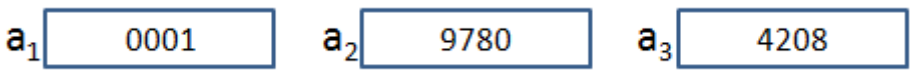
La primera línea del código (`a1=new A();`) crea en el heap (en la dirección de memoria 0001, por ejemplo) una abstracción de un objeto A, con su variable y sus métodos.



Acto seguido se crean otros dos objetos más.



En las variables entonces estarán las direcciones de cada objeto.



Entonces llega la parte donde se ejecutan los sets. El set inserta en la variable x de cada objeto lo que se le pase a la función. Al imprimir se imprime dicho número por pantalla.

Lo interesante en esta parte es el orden en que aparecen las sentencias de asignación:

```
a1 = a2
a3 = a2
```

Por tanto, las direcciones de memoria de a_1 y a_3 son iguales que la de a_2. Por ello, cuando se hace el set de a_3 es el único que importa, pues los anteriores se escriben en el mismo sitio.

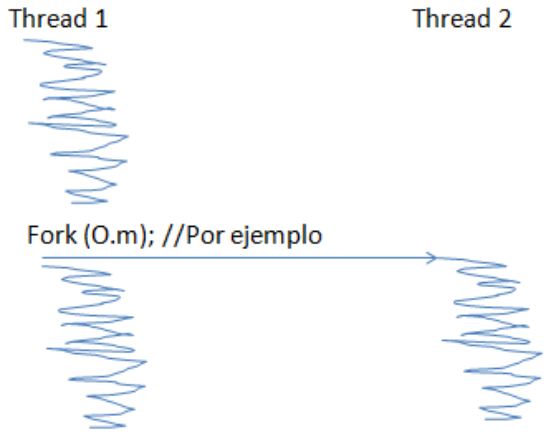
Conceptos

Hilo de ejecución (Threads o procesos)

En concurrencia trabajaremos con muchos hilos de ejecución. Un thread o proceso es lo mismo en con concurrencia. Para poder disponer de varios hilos de ejecución existen varias técnicas, nosotros vamos a utilizar el concepto de Fork.

Fork

Conceptualmente es un punto en el que se crea un nuevo hilo de ejecución que ejecuta de forma simultánea.



El fork del método ejemplo ejecuta ese método en un nuevo thread. Este thread tiene un entorno de ejecución diferente, con variables locales diferentes, etc. Ambos threads solo comparten una cosa: la memoria. Las variables globales son iguales para ambos.

Es decir, si tratas de asignar un valor a una variable global se asignarán a la vez. Es decir, no sabrá qué proceso ejecuta primero ya que entra en juego la sensibilización de la memoria, solo puede ser accedida una vez al tiempo, pero no sabes cuál va a ejecutarse primero. Ese es uno de los problemas de la concurrencia.

Sincronización

En muchas ocasiones será necesarios sincronizar la ejecución de varios threads, esto es algo que dista de ser trivial.

Semáforos

Es un tipo abstracto de datos muy útil para sincronizar procesos:

```
interface Semaphore{
     void Wait();// Decrementa el contado si no es 0, si lo es
                 // mantiene en espera al proceso hasta que deje
                 // de serlo.
     void Signal();// Incrementa el contado
}
```

Los semáforos se encuentran en los sistemas operativos.

Definición de concurrencia

- Ejecución simultánea (threads).

- Interacción entre threads:
  - **Sincronización**. Uno de ellos, probablemente, tendrá que esperar al otro.
  - **Comunicación**. Necesitamos algún sitio para comunicar threads.
    - **Memoria compartida**. Dejar el dato en una variable compartida.
    - **Paso de mensajes**. Deja un mensaje al otro proceso.
    - En esta asignatura no pueden mezclase ambas herramientas, pero en el mundo real puede hacerse.

Concurrencia en Java

¿Cómo pueden establecerse threads en Java?

Clase Thread

Existe una clase predefinida en Java llamada Thread, con tres operaciones:

- start()
- join()
- run()

Ejemplo

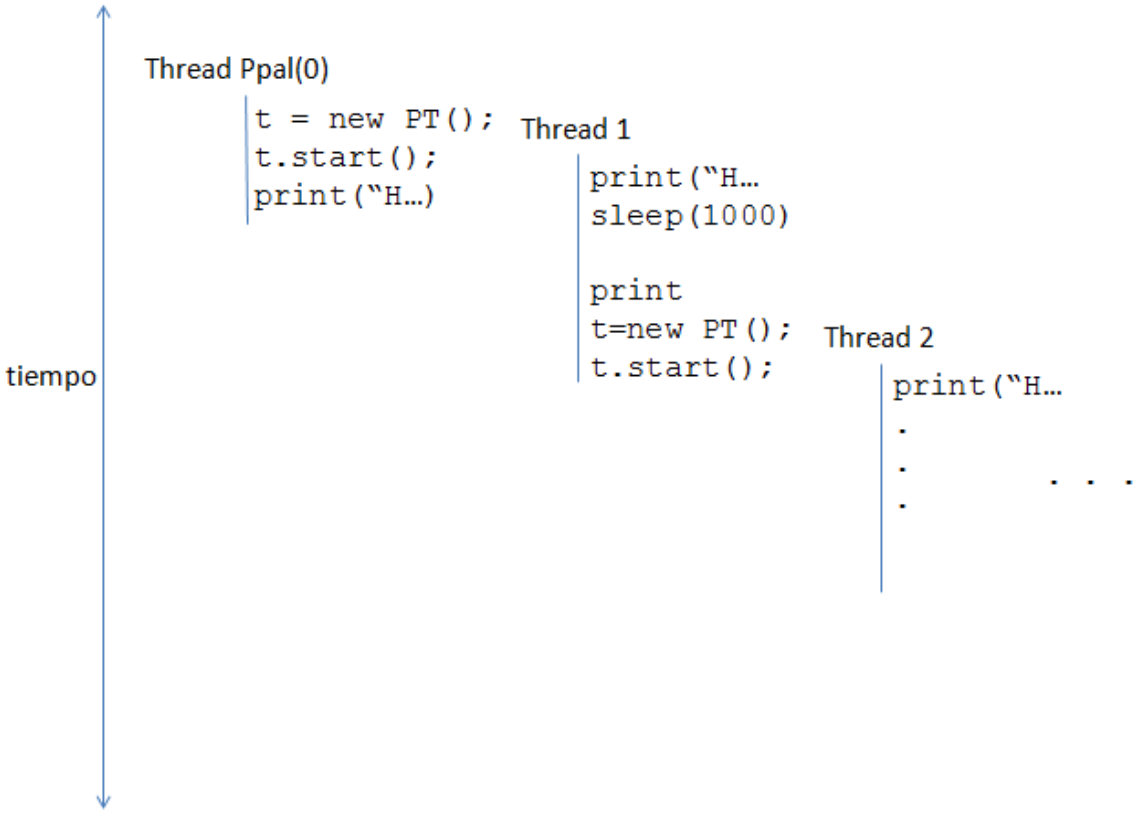
```
void main(){
      Thread t = new PrimerThread(); //Crea un objeto de la clase
                                     //PrimerThread
      t.start(); //start es el método más importante. Lanza un
                 //nuevo proceso en tiempo despreciable y ejecuta
                 //el método run() del thread al que se llama.(1)
      System.out.println("Hola soy el thread principal");
      System.out.println("He puesto en marcha un proceso");
      t.join();//Este método espera a que el hilo de ejecución
               //llamada (t) termina para continuar su propia
               //ejecución.
}

class PrimerThread extends Thread{
      public void run(){//Este run es el main del segundo proceso
            System.out.println("Hola soy un PrimerThread y acabo
de arrancar");
            sleep(1000);//espera 1 segundo
            System.out.println("Hola soy un PrimerThread y voy a
terminar");
      }
}
```

(1) Si en vez de t.start() hubiésemos colocado t.run() entonces no se crearía un proceso nuevo sino que continuaría de forma secuencial. Es decir, ejecutaría run() y, después, ejecutaría la línea después de main. Con start() aseguramos cual se ejecuta primero y donde.

Si en la clase PrimerThread, añadiremos las líneas;

```
      Thread t = new PrimerThread();
      t.start();
```

Entonces crearíamos un bucle infinito de creación de threads. Visualmente sería:



Por tanto, una clase Thread puede crear hilos de ejecución también.

Ejercicio 1

Creación de threads en java

```
public class CC_01_Threads{
      public static void main(String args[])
      {
            int N=1;
            Thread t[] = new Thread[N]; //Crea un array de thread

            for(int i = 0; i < N; i++){
                  t[i] = new PrimerThread; //Crea los elementos de
                                           //dicho array.
            }
            for(int i = 0; i < N; i++){
                  t[i].start(); //Pone en marcha todos los threads
            }
            System.out.println("Todos los procesos en marcha");
            for(int i = 0; i < N; i++){
                  t[i].join(); //Espera a que terminen todos los
                               //procesos.
            }
            System.out.println("Todos los procesos terminados");
      }
}
```

En el run() de la clase thread se escriben cosas y se espera. Por mucho que aumentemos N el programa no tarda mucho más que el tiempo de espera de un thread. Eso es concurrencia.

Ejercicio 2

Provocar una condición de carrera

```
public class CC_02_carrera
{
      public static void main(String [] argv) threads Exception
      {
            final int M=2;
            Thread t[] = new Thread[M];
            for(int i = 0; i < M; i++){
                  t[i] = new Incrementador;
            }
            for(int i = 0; i < M; i++){
                  t[i].start();
            }
            for(int i = 0; i < M;  i++){
                  t[i].join();
            }
            System.out.println("Incrementador.n");
      }

      Class Incrementador extends Thread
      {
            final private static int N = 10;
            static int n=0;
            public void run()
            {
                  for(int i = 0; i < N; i++){
                        n++;
            }
      }
}
```

IMPORTANTE
Si no es static cada objeto tendría su propia n

Otra forma de compartir n sería creando una clase para ello:

```
public class Entero
{
      public int = 0;
}
```

y creando un objeto de esta clase en el main:

```
Entero compartido = new Entero();
```

y se construye el constructor en Incrementador:

```
public Incrementador(Entero compartido)
{
      this.compartido = compartido;
}
```

aumentarlo en cada iteración:

```
compartido.n++
```

y, finalmente, al imprimirlo:

```
System.out.println(compartido.n)
```

Al ejecutar este programa se observa que no siempre se obtiene lo que se debe ya que existen iteraciones en las que no se incrementa la variable. Esto sucede porque existen condiciones de carrera, es decir, porque dos procesos han intentado acceder a la misma variable al mismo tiempo. Como un acceso a memoria no puede hacerse de forma concurrente, primero lo hará uno proceso y después otro, haciendo que no seamos capaces de conocer de forma precisa el resultado final.

Volatile

Esta palabra reservada de Java crea una barrera para la caché que impide (conceptualmente) que se utilicen dos variables (idénticas) en dos cachés diferentes, técnica que se utiliza en los procesadores multicore donde cada core tiene una caché diferente.

Ejercicio 3

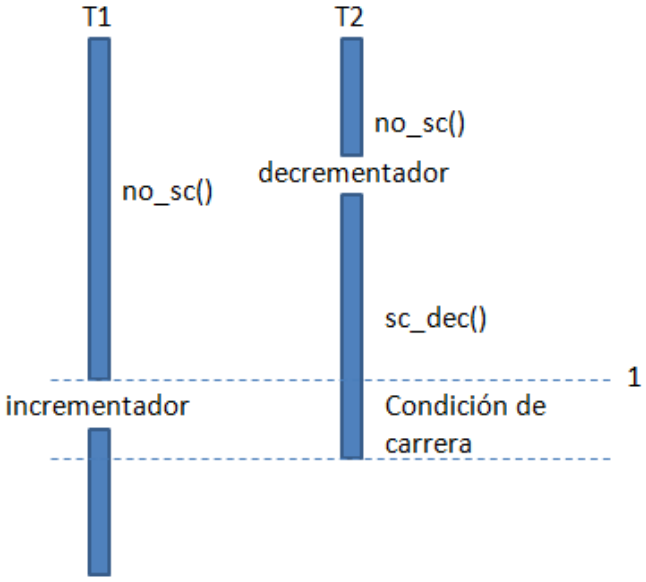
Garantizar exclusión mutua con espera activa

```
class CC_03_MutexEA{
	//número de veces que los procesos repiten su labor
	static final int nPasos = 10000;

	//variable compartida
	volatile static int n = 0;

	//sección no crítica
	static void no_sc(){
	...
	}

	static void sc_inc(){
		n++;
	}

	static void sc_dec(){
		n--;
	}
}
```

La sección crítica de cada proceso no puede ser ejecutada al mismo tiempo:

```
static class Incrementador extends Thread{
	public void run(){
		for(int i = 0; i < nPasos; i++){
			//Sección no crítica
			no_sc();
			//Sección crítica
			sc_inc();
```

```
            }
        }
}
```



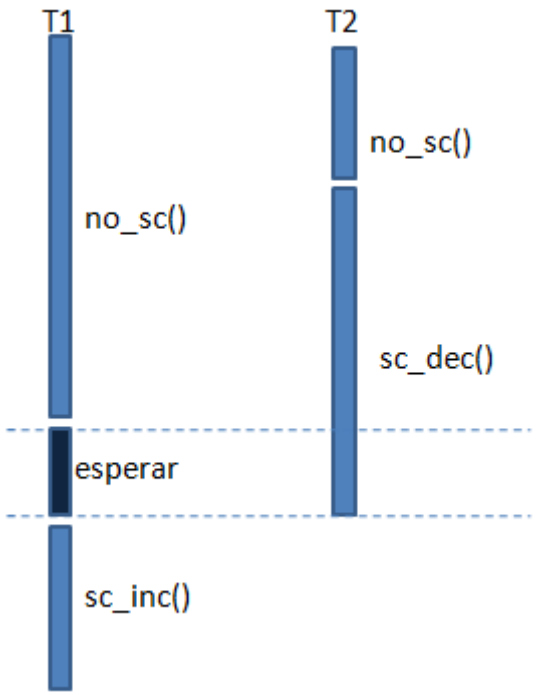
¿Qué va a suceder?

En el instante de tiempo 1 (imagen de la izquierda) el incrementador ha terminado de ejecutar no_sc() y va a ejecutar sc_inc(), pero si se lo permitimos durante el tiempo en que ambas secciones críticas ejecutan a la vez puede darse una condición de carrera. Lo que debemos conseguir es que un proceso espere a otro. Sin embargo, debe hacerse sin herramientas de concurrencia (en este ejercicio), solo mediante control de flujo y espera activa.

Básicamente un while:

```
no_sc();
while(dec no ejecuta sc){
      sc_inc();
}
```

Por tanto el esquema de lo que debe suceder será algo diferente.



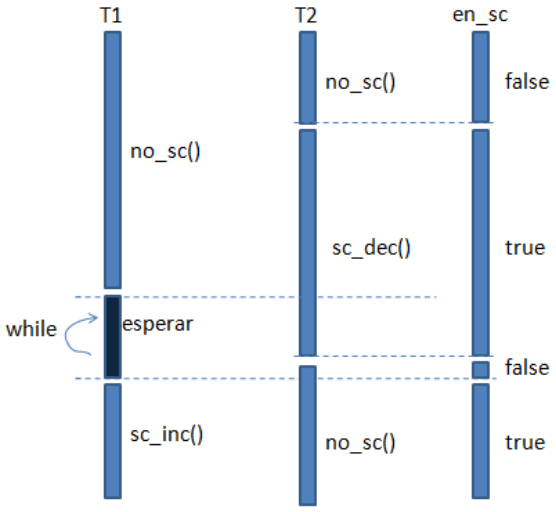
Para llegar a esta solución una idea plausible sería tomar una variable booleana:

```
//variable para asegurar mutex
volatile static boolean en_sc = false;
//si nadie está ejecutando está a false
```

Es muy importante la etiqueta volatile. Ahora el while pasaría a tener un formato similar a:

```
while(en_sc){}//comprueba si alguien está ejecutando la sc
              //y espera
en_sc = true;
sc_inc();
en_sc = false;
}
```

Esto parece una solución (en el diagrama de la derecha), pero no lo es. Ya que si los dos entran al mismo tiempo en el while se dará una condición de carrera de todas formas.



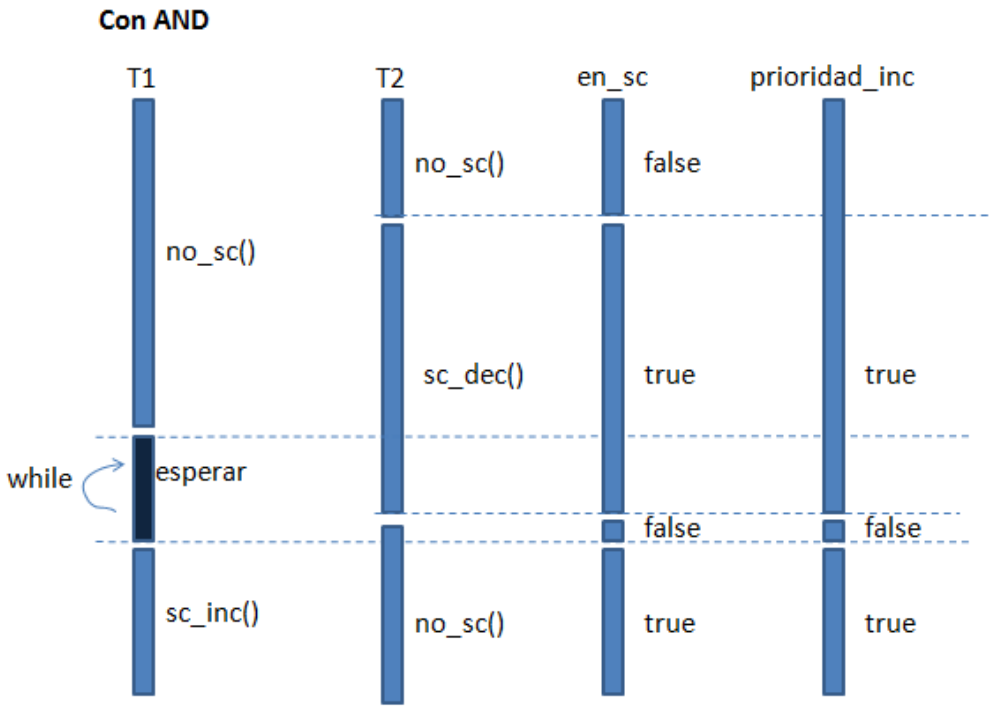
Otra idea es añadir una variable compartida más que establezca prioridad a los procesos. Entonces:

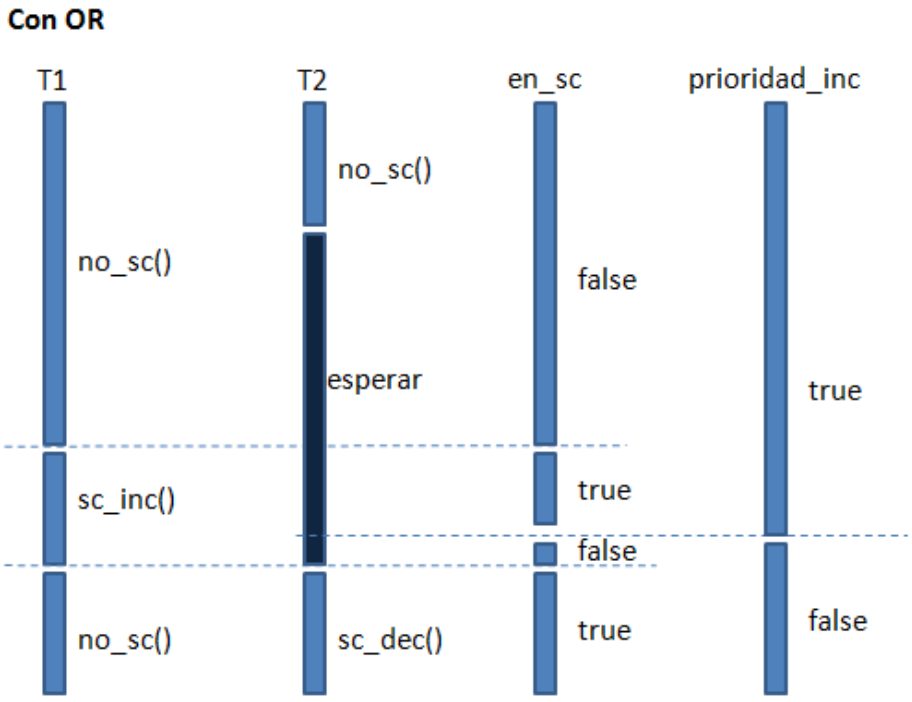
```
volatile static boolean prioridad_inc = true; //Prioridad para
                                              //el incrementador
```

INC
```
//esperar si no tengo la prioridad y en_sc
while(!prioridad_inc || en_sc){}
en_sc = true;
sc_inc();
prioridad_inc = false;
en_sc = false;
```

DEC
```
//esperar si no tengo la prioridad y en_sc
while(prioridad_inc || en_sc){}
en_sc = true;
sc_dec();
prioridad_inc = true;
en_sc = false;
```

Tendríamos dos diagramas diferentes si en la condición del while estableciésemos un AND en lugar de un OR.





Sin embargo, no tendremos que preocuparnos de ello puesto que no es una solución válida.

Otra idea es la de crear dos variables booleanas. Entonces:

```
volatile static boolean en_sc_inc = false; //cierto cuando inc
                                           //entra en sc
volatile static boolean en_sc_dec = false; //cierto cuando dec
                                           //entra en sc
```

```
en_sc_inc = true;
while(en_sc_dec){}//si dec está en sc, inc espera
sc_inc();
en_sc_inc = false;

[y de forma simétrica en el dec]

en_sc_dec = true;
while(en_sc_inc){}
sc_dec();
en_sc_dec = false;
```

Esta es una solución inválida, también, ya que si se ejecutan los dos procesos al mismo tiempo, ninguno entrará en la sc, a esto se le denomina interbloqueo o deadlock.

Una última solución podría ser utilizar una variable compartida que se modifique además de las variables independientes. En un método se haga verdad y en el otro falso y sea también argumento para el while. Esto sigue sin ser una solución.

Solución al ejercicio 3

En incrementador

```
en_sc_inc = true;
turno_inc = false;
while(en_sc_dec && !turno_inc){}
sc_inc();
en_sc_inc = false;
```

En decrementador

```
en_sc_dec = true;
turno_inc = true;
while(en_sc_inc && turno_inc){}
sc_dec();
en_sc_dec = false;
```

Propiedades indeseables

- **Deadlock** → Ambos procesos están bloqueados esperando a que el otro deje de estarlo. Ninguno avanza, jamás.
- **Inanición** → Similar al deadlock con la diferencia de que existe una posibilidad de que uno de ellos salga del bloque.

Mecanismos de concurrencia

Semáforos

Es un TAD con dos operaciones atómicas, esperar y señalizar:

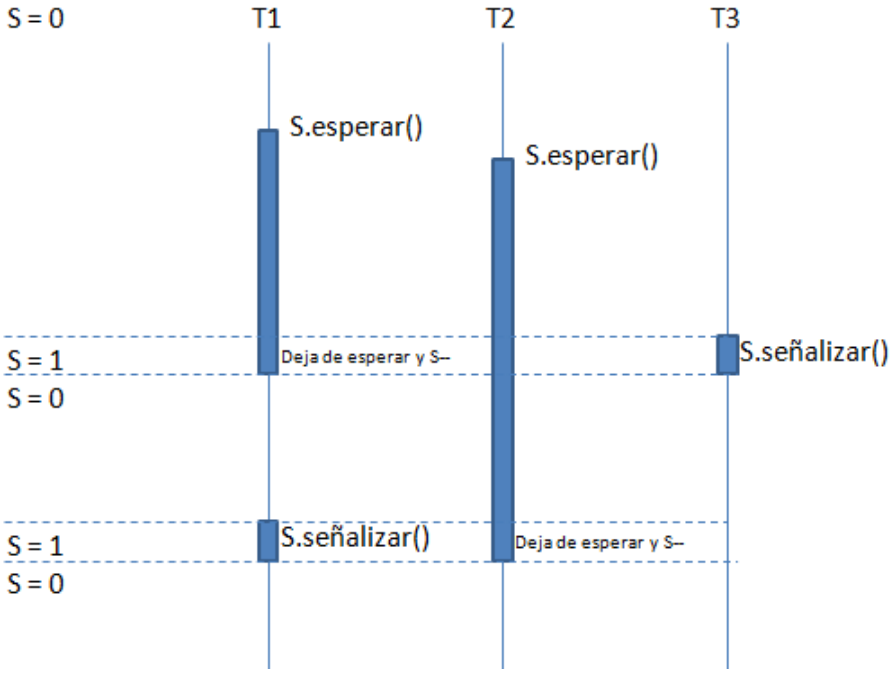
```
class semáforo{
     private int cont = 0; //Contados, a 0 en este ejemplo.

     public void esperar(){
```

```
            /** Espera hasta que cont > 0 y entonces cont-- */
      }

      public void señalizar(){
            cont++; //Incrementa el contador.
      }
}
```

Para evitar condiciones de carrera cada operación de los semáforos está implementada atómicamente, esto quiere decir, que dos procesos no pueden ejecutar al mismo tiempo las operaciones de un semáforo. Si dos procesos tratan de efectuar la misma operación al mismo tiempo la efectuará primero la que llegue primero, sin sorpresas.

La utilidad de un semáforo es que cuando un proceso ejecute esperar() y el contador esté a 0, esperará hasta que otro proceso aumente el contador (al finalizar su sección crítica, por ejemplo).

La cuestión ahora es qué sucede cuando dos o más procesos están esperando y otro proceso ejecuta señalizar. Lo que sucede es que un semáforo solo va a desbloquear a uno de los procesos, uno y solo uno; como puede verse en el diagrama. Siguiendo una propiedad de la concurrencia denominado Fairness (equitatividad).



Existen tres tipos de equitatividad:

- Equitatividad fuerte (FIFO). Garantiza equitatividad en el infinito.
- Equitatividad débil (Azar). Garantiza equitatividad en el infinito.
- Equitatividad “nula” (LIFO). No garantiza equitatividad en el infinito.

Los semáforos en las librerías

Cuando te encuentras ante una librería de semáforos debes mirar dos cosas básicas:

- Siempre están las operaciones esperar y señalizar, hay que mirar como se denominan en esa librería. En Java: esperar = acquire y señalizar = release.
- Hay que mirar cómo se construyen los semáforos. En Java: El constructor se llama con uno o dos parámetros:

- Un contador (int) que será el número inicial del contador del semáforo. Parámetro obligatorio.
- Un parámetro booleano que construirá el semáforo con equitatividad fuerte (si true) o sin equitatividad (a false). Es un parámetro opcional.

En la librería propia de la asignatura, CCLib, tenemos las siguientes características:

- El constructor ya no necesita parámetros (por defecto, int=0 y con equitatividad), aunque pueden darse.
- Esperar es await(). No necesita capturar la excepción y siempre decrementa el contador.
- Señalizar es signal().

Ejercicio 5 y 6

Almacén de uno y varios datos con semáforos

Productor

```
while(true){
      p = fabrica.producir();
      //sc
      almacenCompartido.almacen(p);
      //sc
}
```

Ya hemos aprendido a solucionar la exclusión mutua (mutex), trivial con semáforos (ejercicio 4 del fichero de ejercicios, no se especifica en estos apuntes). El problema realmente complicado ahora es la sincronización condicional. Veamos un ejemplo:

Almacén

```
Class Almacen1 implements Almacen
      //Producto a almacenar: null representa que no hay producto
```

Cuando un consumidor quiere un producto:

```
//mutex == 0 ➔ proceso en sc
//Dato == 1 ➔ almacenado != null
public producto extends ...
      .
      .
      .
      //sc  if ("algo" == null)
      mutex.await();
```

Solución del ejercicio 5 y 6

Puede resolverse con solo dos semáforos (diagrama 1). Sin embargo, esta solución no es significativa pues no siempre puede resolverse así, es necesario (casi siempre) disponer de tres semáforos:

- Uno que permita la exclusión mutua.
- Uno que indique cuando el almacén está vacío.
- Otro que indique cuando el almacén esté lleno.

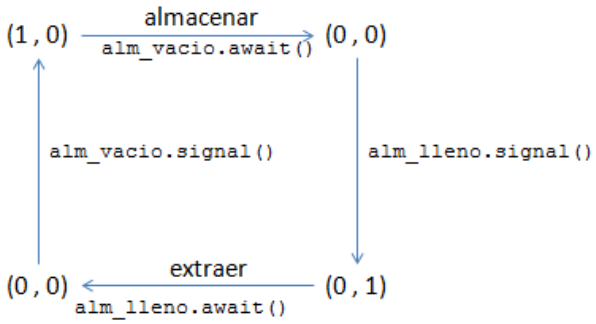
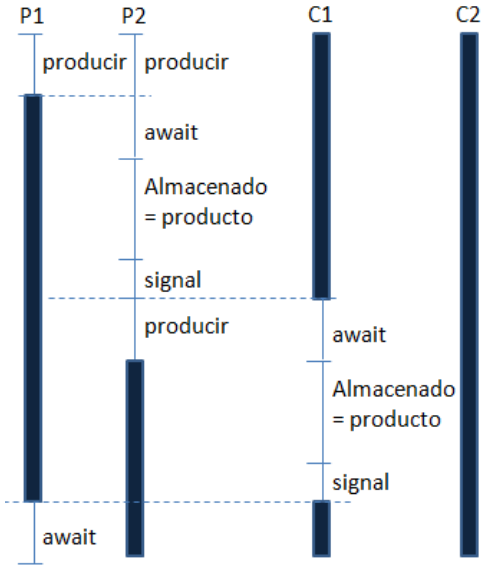


Diagrama 1



Aunque esta no sería la única solución. Podemos añadir un semáforo de mutex sobreprotegiendo la sección crítica. Pero la solución definitiva es la siguiente:

Productor

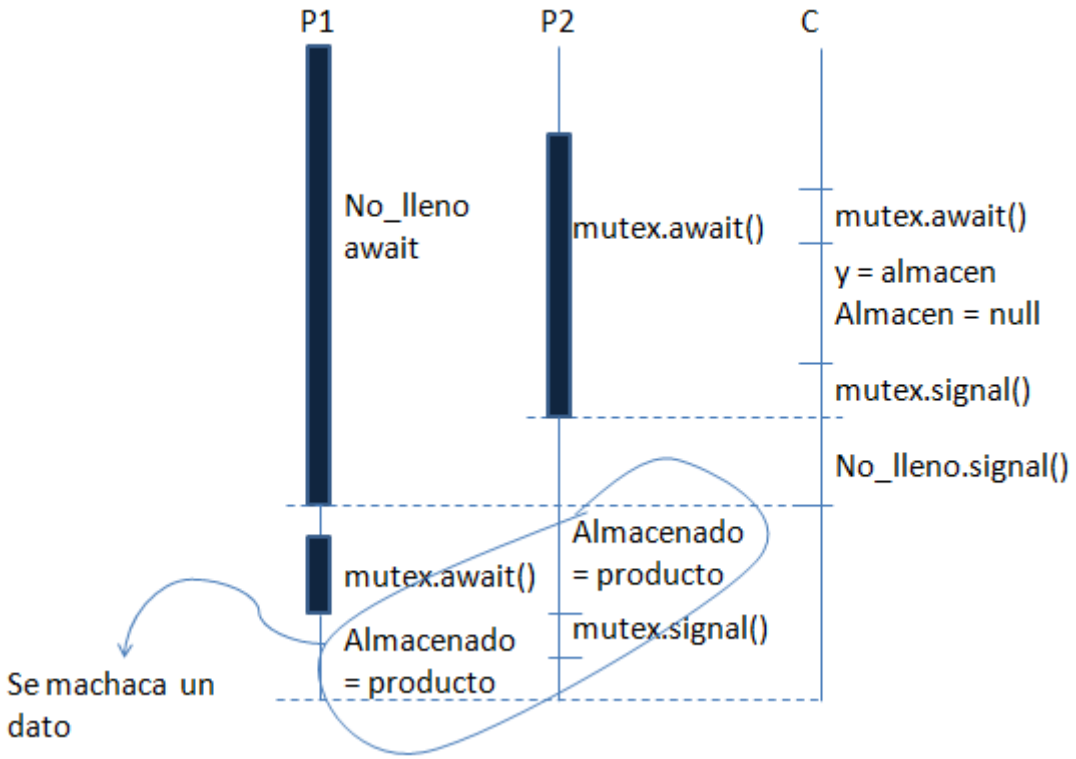
```
mutex.await();
if(almacenado !=null){
      mutex.signal();
      no_lleno.await();
      mutex.await();
}
almacenado = producto;
mutex.signal();
no_vacio.signal();
```

Consumidor

```
mutex.await();
if(almacenado == null){
```

```
        mutex.signal();
        no_vacio.await();
        mutex.await();
}
almacenado = null;
mutex.signal();
no_lleno.signal();
```

Este ejemplo es muy difícil de encontrar debido a la dificultad de garantizar la solución correcta con semáforos. Además, no es la solución más óptima, pues si almacenado ¡ = null:



Resumen hasta ahora

Los procesos **deben pararse** cuando algo *no está bien* y deben volver a **ponerse en marcha** cuando está *bien*.

Razones para pararse

- **Mutex.** Ej: un proceso quiere almacenar y otro proceso está almacenando o extrayendo.
- **Sincronización condicional.** La estructura de datos está en el estado apropiado para realizar una operación. Ej: un proceso quiere almacenar y el buffer está lleno.

¿Qué hay que hacer?

El proceso tiene que bloquearse cuando las condiciones no son apropiadas y otro proceso tiene que desbloquearlo cuando las condiciones sean propicias (para no consumir CPU durante la espera).

¿Qué no hay hacer?

- Espera activa (¿puedo?¿puedo?¿Puedo?...).

- Jamás ejecutar la operación si la condición no se cumple.
- Jamás dejar de ejecutar la operación (hay que esperar a que se cumpla la condición.

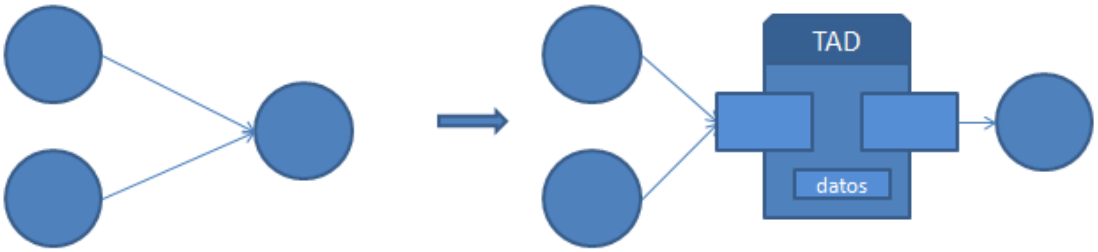
Invariante → Propiedad del sistema que quiero que se mantenga siempre.

```
Semaphore alm_lleno
//Invariante: alm_lleno == 1 ⇔ almacenado != null
//            Λ (alm_lleno == 1 Λ alm_lleno == 0)
//            Λ alm_lleno == 0 ⇔ almacenado == null
```

Especificación de recursos

Resumen de los apuntes oficiales

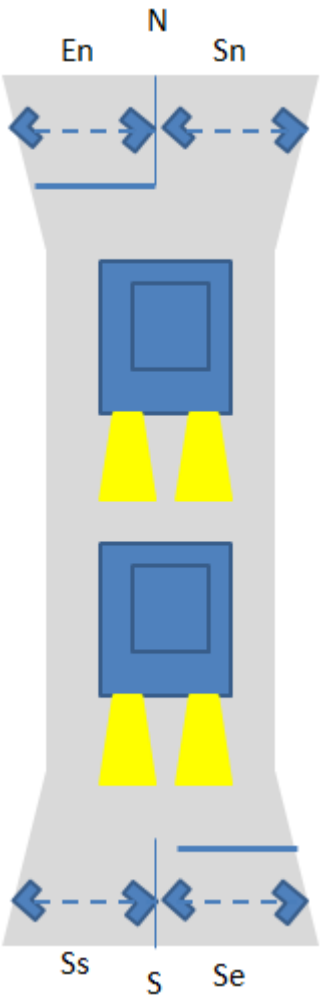
- Idea general: procesos y recursos para comunicarse.
- Procesos: código.
- Recursos:
  - Nombre (como en una clase).
  - Operaciones y datos (como en una clase)
  - Semántica.
- Sobre nuestro ejemplo:
  - Semántica
    - Datos.
    - Mutex (“gratis”).
    - Sincronización condicional (CPRE).
    - Operaciones: PRE-POST (siguen reglas lógicas).



La comunicación director puede provocar problemas. → Creamos un TAD que sea un recurso compartido protegido que permita la concurrencia.

Ejercicio 7

Especificación de un recurso compartido



C-TAD ControlAccesoPuente

Operaciones

ACCION Solicitar_entrada: Entrada[e]
ACCION Avisar_salida: Salida[e]

SEMANTICA

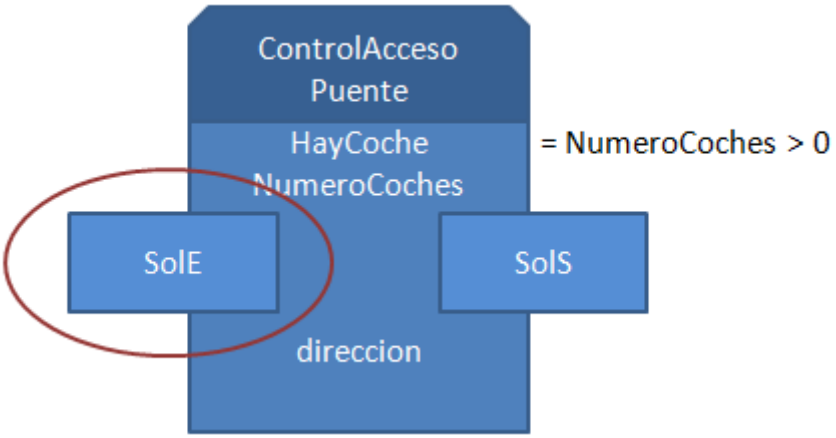
DOMINIO

TIPO: ControlAccesoPuente = (sn: N x ns: N)
TIPO: *Un controlador para los coches que van sur-norte y otro para los que van norte-sur. Uno de ellos siempre debe ser 0.*
Donde: Entrada = EN/ES
Salida = SN/SS

INVARIANTE: self.sn > 0 ➔ self.ns
Λ self.ns > 0 ➔ self.sn

CPRE: *Si se entra por el sur no puede haber coches norte-sur, si se entra por el norte no puede haber coches sur-norte.*
PRE: e = ES Λ self.ns = 0 V e = EN Λ self.sn = 0
Solicitar_entrada(e)
POST: e = ES Λ self.sn = $self.sn^{PRE}$ +1 Λ self.ns = $self.ns^{PRE}$
V e = EN Λ self.ns = $self\ ns^{PRE}$ + 1 Λ self.sn = $self.sn^{PRE}$

CPRE: *Es imposible que esta operación sea ejecutada más de una vez al tiempo por culpa de un coche que no esté cruzando el puente, por lo que siempre puede ser ejecutada.*
CPRE: Cierto
avisar_salida(s)
POST: *Decrementamos el número de coches en la dirección adecuada.*
POST: S = SS Λ self.sn = $self.sn^{PRE}$ ➔ self.ns = $self.ns^{PRE}$ -1
V S=SN Λ self.ns = $self.ns^{PRE}$ ➔ self.sn = $self.sn^{PRE}$ -1



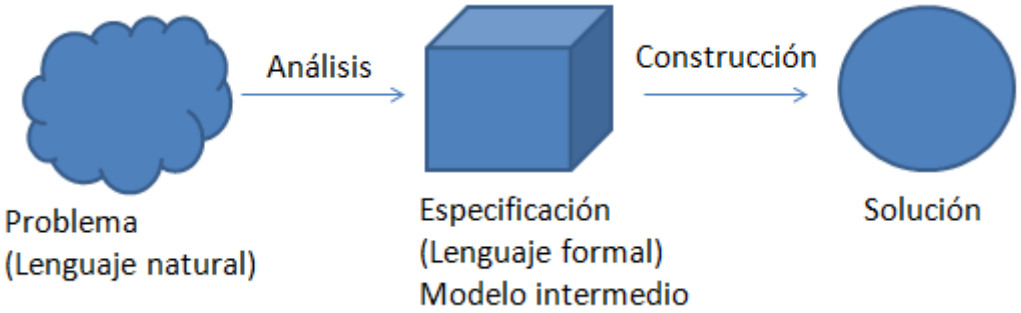
SolE para si:

HayCoches Λ dirección contraria ➔¬HayCoches V "misma" dirección

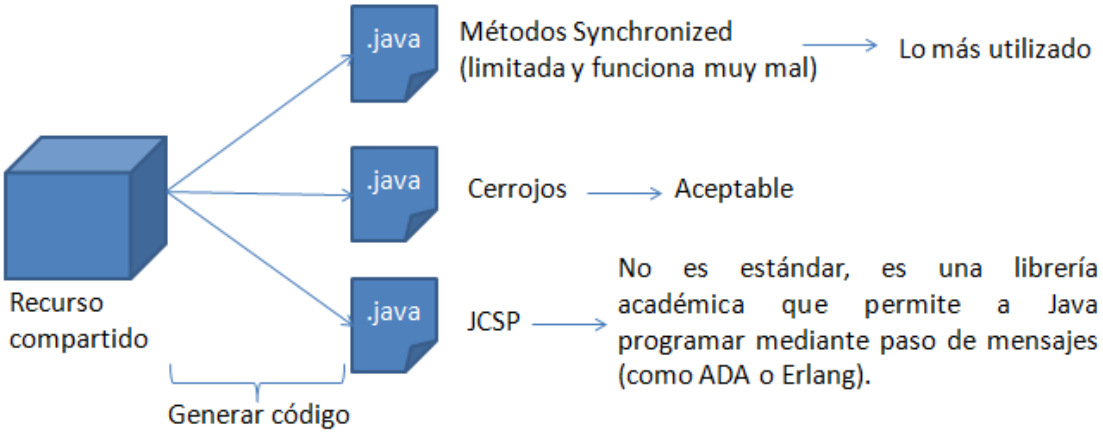
Tema 2

Algunos conceptos iniciales:

- MDE – Ingeniería basada en modelos



  - En nuestro caso:



Métodos Synchronized

Recomendamos la lectura del capítulo 17, secciones 17.1 y 17.8 del documento: *The Java Language Specification, Third Edition*.

- Concurrencia = ejecución simultánea (no determinista) + interacción
- Interacción = comunicación + sincronización
- Sincronización = Exclusión mutua + por condición

Tendremos un modificador nuevos para nuestras variables y métodos, Synchronized, que nos permitirá tener **exclusión mutua** implícita. Este modificador impide que más de un thread tenga acceso al mismo tiempo al recurso compartido.

Lo complicado es dotar a nuestro sistema de sincronización por condición. El bloqueo es explícito mediante el método wait(). Es desbloqueo también es explícito pero existe un problema: no es selectivo. Es decir, somos incapaces de decidir a quién desbloqueamos. Este método es notifyAll(), que envía una señal a todos los threads en espera y uno deja de estarlo y pasa a estar ejecutable(de forma indeterminada).

Estos métodos Synchronized son muy sencillos pero terriblemente ineficientes.

Ejemplo

```
Class Almacen1 implements Almacen{
      private Producto almacenado = null;
      private boolean hayDato = false;


      public Almacen1(){ //Constructor vacío
      }

      public synchronized void almacenar(Producto producto){
            //Protocolo de acceso a la sección crítica y código
            //de sincronización para poder almacenar
            while(hayDato){
                  try{
                        wait();
                  }catch(…){}
            } //Aquí se cumple mi CPRE, se cumple not hayDato
              // ➔ debido a que es un while.
            //Sección crítica
                  almacenado = producto;
                  hayDato = true;

            //Protocolo de salida de la sección crítica y código
            //de sincronización para poder extraer
            notifyAll();
      }

      public synchronized Producto extraer(){
            Producto result;
            //Protocolo de acceso a la sección crítica y código
            //de sincronización para poder extraer.
            while(!hayDato){
                  try{
                        wait();
                  }catch(…){}
            }
            result = almacenado;
            almacenado = null;
            hayDato = false;
            notifyAll();
            return result;
      }
}
```

El ejercicio 8 es muy similar a este, solo que los productores almacenan M datos y los consumidores extraen N datos. Tenemos un ejemplo de un multibuffer como esté en el artículo sobre especificación.

Dos cuestiones importantes:

- Imposible poner if-then para entrar en el wait.
- La CPRE depende de un valor externo, el número de datos (parámetro de la función) que se quieren almacenar o extraer.

Cerrojos (Locks & conditions)

Antes de nada se recomienda:

- Leer el capítulo 3.4 en Andrews & Schneider, *Concepts and Notations for Concurrent Programming*.
- Leer el artículo de Brinch Hanser, *Java's Insecure Parallelism.*
- Leer la documentación sobre locks & conditions en el tutorial de concurrencia de Oracle.

Los métodos synchronized nos daban la exclusión mutua de forma implícita. En cerrojos, la exclusión mutua es explícita, hay que programarla. Para entrar en exclusión mutua debemos ejecutar:

```
mutex.lock(); → Mutex:ReentrantLock
```

y para salir de ella:

```
mutex.unlock();
```

Los cerrojos nos permiten decidir en qué cola ponemos cada thread (toman esta idea del concepto de monitor). Estas colas se denominan *queue conditions* o simplemente, *conditions*. Así puedes poner threads similares en la misma cola sin que se mezclen entre sí.

Crearíamos la cola mediante:

```
cextraer = mutex.newCondition();
```

Se introduce un thread en la cola con la instrucción:

```
cextraer.await();
```

y se extrae de la cola con:

```
cextraer.signal();
```

En Java, para trabajar con cerrojos utilizaremos el paquete *java.util.concurrent.locks.* Concretamente las interfaces:

- Lock → Que encapsula la idea de monitor y controla la exclusión mutua.
- Condition → Que sirve para implementar el concepto de condition queues de los monitores.

y las clases:

- ReentrantLock → Que nos permite anidar cerrojos (que desde la invención de los monitores se sabe que es algo intrínsecamente malo y debe prohibirse). No vamos a utilizar todos sus métodos por este motivo.

Para otorgar exclusión mutua a un objeto lock:

```
Lock mutex;
mutex = new ReentrantLock();
```

```
[…]
mutex.lock() //Para tomar un recurso en exclusión mutua si ya
.            //hay alguien haciendo uso del recurso se mantiene
.            //en espera en alguna de las colas.
.
mutex.unlock()//Se libera el recurso.
```

Para poder garantizar la sincronización por condición con esta idea necesitamos hacer que los objetos lock conozcan la existencia de los objetos condition puesto que son estos los que van a enviar y recibir threads hacia y desde las colas de condiciones. Por ello, en Java, siempre que se quiera crear una condition debe hacerse desde el método para tal efecto del objeto lock:

```
ci = mutex.newCondition();
```

Por eso, cuando se ejecuta el await se sabe que se refiere al objeto cerrojo que lo creó:

```
mutex.lock();
if ¬CPRE
     ci.await();
.
.    //Código de la OP
.
cj.signal(); //Código de desbloque
mutex.unlock();
```

Ejemplo

```
Class Almacen1Locks implements Almacen{
     // Declaración de variables almacenado y hayDato
     // Declaración de cerrojos y condiciones
     Lock mutex;
     Condition cAlmacenar; // Condition empieza con c minúscula
     Condition cExtraer;

     public Almacen1Locks(){
          mutex = new ReentrantLock();
          cAlmacenar = mutex.newCondition();
          cExtraer = mutex.newCondition();
     }

     public void almacenar(Producto producto){
          mutex.lock(); //Acceso a sc

          // Código de sincronización.
          if(hayDato){
               try{
                    cAlmacenar.await();
               } catch(InterruptedException e){}
          }
          // Asegurar la CPRE: se debe cumplir not hayDato!!!
          // sc
          almacenado = producto;
          hayDato = true;
          // Se cumple la CPRE de extraer.
          //Por eso podemos salir de sc y permite ejecutar
```

```
            // un extraer.
            cExtraer.signal()
            mutex.unlock()
        }
[...] // y de forma simétrica en extraer.
}
```

De forma intuitiva hay que garantizar que entre un signal y un await no pueda colarse nada.

La solución más satisfactoria es tener vectores de conditions haciendo que se pare y se añada a la condition de su parámetro de entrada. Es decir, tendremos capacidad/2 condition cAlmacen y cExtraer.

Política 0,1

0 ó 1 await (en un if).

0 signal si no existe ningún thread en espera y, si existen varios threads en espera, seleccionar uno de ellos y confiar en que él permita desbloquear a otro threads como él.

El desbloqueo solidario es el que desbloquea threads del mismo método (cAlmacenar en Almacenar, por ejemplo) para que continúen garantizando que se cumple la CPRE de cada thread llamado y que no existe inanición.

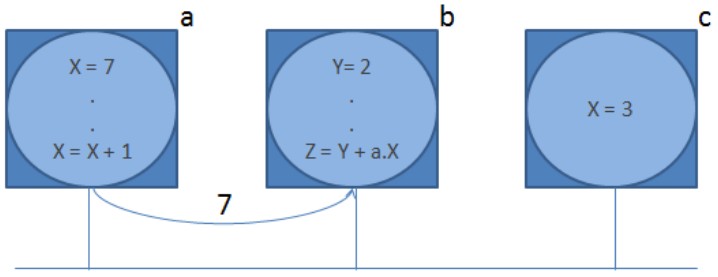
Notas para la práctica

Problemas de Locks & Conditions con Java: el thread que ejecuta el signal se coloca al final de la cola de locks del mutex. Eso permite que entre la llamada al signal y el despertar del thread se cuele otro thread que cambie la CPRE. Eso no debería suceder.

Para evitar este problema se ha creado una librería en cclib llamada Monitor.java que se utiliza exactamente igual que locks & conditions. Sin embargo tiene algunas diferencias sintácticas:

- Clase Lock → Clase Monitor
- Método lock → Método monitor
- Método unlock → Método leave
- Clase Condition → Clase Cond

Concurrencia basada en paso de mensajes



Pasamos de un contexto de compartición de memoria a un contexto de sistema distribuido. Ningún proceso tiene acceso directo a los procesos de otras máquinas. Si el proceso b desea sumar su variable con la variable x del proceso a, no puede hacerlo directamente.

Lo que vamos a estudiar en este tema son las distintas formas que existen para comunicar máquinas una con otra.

Java no es el lenguaje más indicado para sistemas distribuidos (vaya, qué le vamos a hacer). Otros lenguajes mejores en estos casos son:

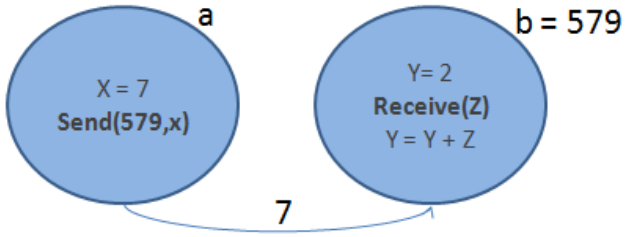
- ADA 95 → Usado en aviónica, defensa, tecnología espacial… En sistemas críticos y altamente distribuido.
- Erlang → Usado, en principio, para terminales digitales. Ha terminado siendo utilizado para aplicaciones de envío de mensajes en internet y aplicaciones masivas concurrentes: VoIP, Twitter, Whatsapp, Tuenti, Facebook…

Todos estos lenguajes se basan en la arquitectura teórica CSP [Section 4, A+S] (Communicating Sequential Processes) para el paso de mensajes.

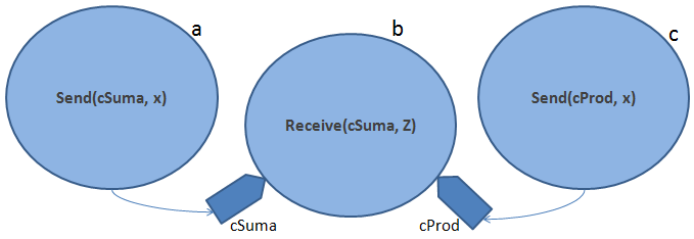
En paso de mensajes no existe la posibilidad de compartir variables, se elimina el problema de la exclusión mutua. El envío de mensajes va a ser el encargado de la sincronización.

El primer problema que debemos resolver es el direccionamiento. En un sistema de direccionamiento ambas estaciones (emisor y receptor) deben ponerse de acuerdo sobre cómo van a comunicarse. Existen varias formas:

- Puertos.
- Identificador de proceso (más primitivo, más bajo nivel). Un sistema muy poco flexible. No es una buena idea.

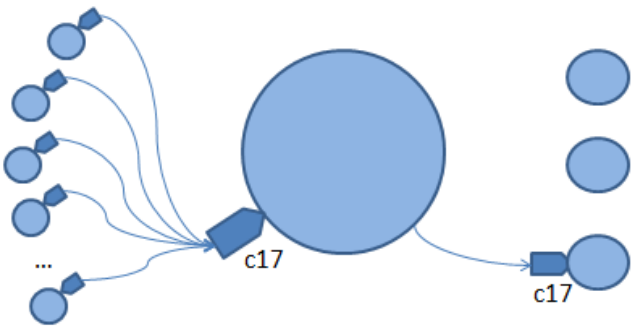


- Canales (una abstracción de la idea de puerto). Un canal es un identificador conocido por todos los procesos que son capaces de determinar de quién y por donde envían sus mensajes.




```
cSuma.send(x)
.
.
cSuma.receive(z)
```

Esta técnica es capaz de delegar el paso de un mensaje a otros procesos:

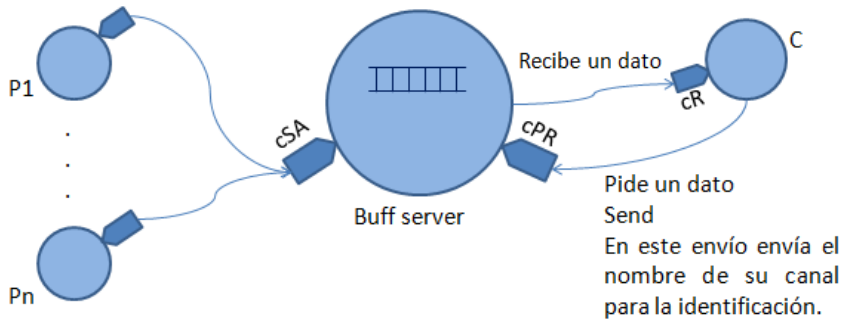


Existen diferentes tipos de canal:

- Según su aridad:
  - 1 : 1 (caso de los servidores)
  - n : 1 (el más común)
  - n ; m (propuesta teórica poco utilizada)
- Según su sincronía:
  - Asíncrono: El envío no bloquea, después de enviar el emisor podría seguir ejecutando antes de que llegue la respuesta.
  - Síncrono: El envío bloquea, obliga a que emisor y receptor estén sincronizados. El send va a bloquear hasta que el receptor ejecute una sentencia receive. Es más cómodo que un paso de mensajes asíncrono.

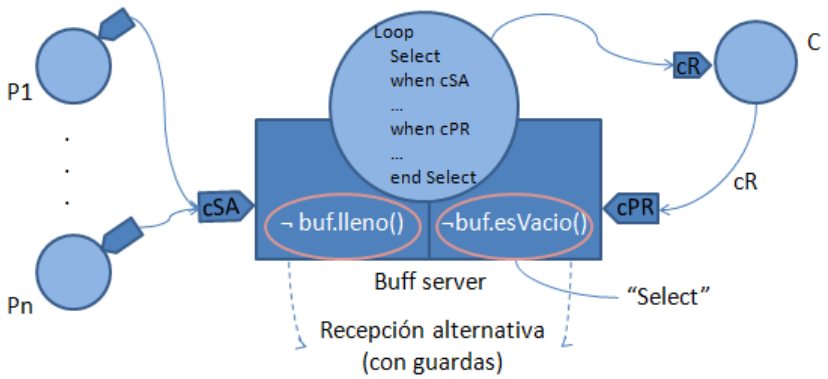
En paso de mensajes cada thread tiene su espacio de memoria. No existen las variables compartidas, lo único que se comparte es el espacio de direcciones.

En un sistema con canales (n : 1) Send puede o no ser bloqueante. Si Send es bloqueante (receive siempre lo es) estaremos ante un paso de mensajes síncrono.



En el siguiente ejemplo el servidor posee un buffer que llenará con lo enviado por los procesos productores y los procesos consumidores lo vacían. Cuando el buffer está vacío parece evidente que no se quiera recibir por cPR porque bloquearíamos cSA y no podríamos recibir datos. Por ello siempre se comienza con un Send de los productores. El problema llega cuando el buffer tiene uno o más datos (pero no está lleno) puesto que si atiende la petición del consumidor bloquea el recibo de la información de los productores y viceversa. Esto destruye la motivación del problema.

Convertir dos canales bloqueantes, como los vistos anteriormente, en un sistema que alterne peticiones es la motivación principal del CSP. De forma gráfica lo veríamos como:

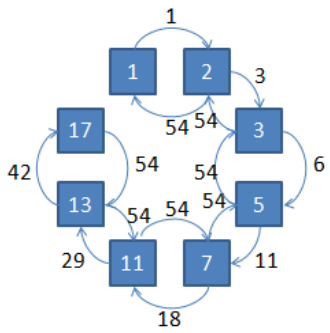


Si atendemos varios receives y el buffer se vacía hay que avisar a cR. Puede ocurrir lo contrario, se puede llenar la cola y, entonces, no podremos almacenar las peticiones cSA. Para resolver este problema disponemos del concepto de CSP que es la **recepción alternativa con guardas.** Este sistema dotará al sistema de dos booleanos que activarán y desactivarán los canales según la condición. El código del bucle cambiará ligeramente:

```
loop
      select
      when ¬buf.lleno()
            buf.push(cSA, receive());
            …
      when cPR
            …
      end select
```

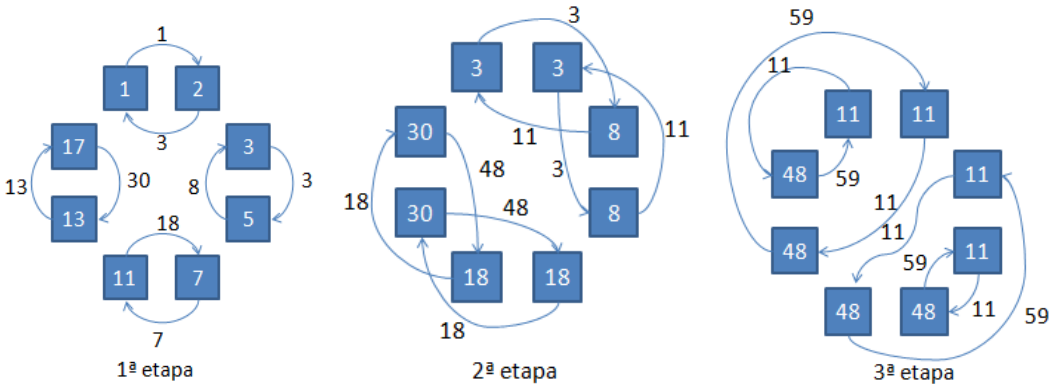
Ejemplo de algoritmo distribuido

Nuestro objetivo será sumar todos los números del sistema y que todos los procesos conozcan el número final. La solución gráfica mediante un sistema lineal sería:



Que se resuelve con 14 etapas.

Utilizando un sistema distribuido:



Tan solo se necesitan 3 etapas, lo cual es una excelente mejora.

CSP en Java: JCSP

Clases necesarias para la creación de procesos. No se hace como lo hemos hecho hasta ahora, tiene su propia jerarquía de clases para JCSP. Se utiliza la interfaz CSProcess implementando

run(). El método run() de CSProcess se comporta igual que en threads, sin embargo, para activar dichos procesos no se hace un for de start() como en threads, sino que se activan utilizando la clase Parallel y llamando al servicio "new Parallel(…)".

Para crear canales se utiliza la clase channel (dependiendo de la llamada hacemos N:1 o 1:1):

- `channel.any2one(); //N:1`
- `channel.one2one(); //1:1`

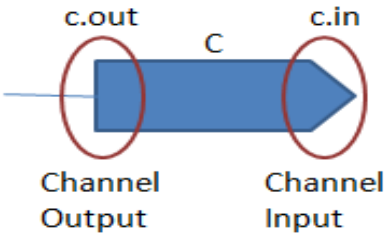
Dicho canal tiene una entrada por cada lado que se crean:

- `Channel Output //Para la entrada.`
- `Channel Input //Para la salida.`

Hecho esto podremos referirnos:

- `c.out();`
- `c.in();`

Para poder escribir dentro de un canal.



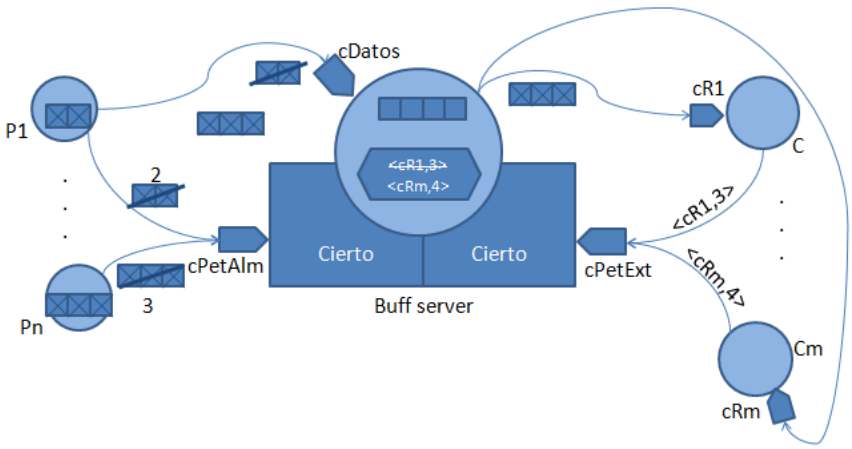
Para crear el resto de nuestro diagrama, primero hacemos la construcción alternativa entre los canales:

```
Servicios = new Alternative ([cAlmacenar,Reltet,inc]);
```

También es necesario crear la conexión entre el bufServer y el proceso C, eso se responde al hacer el servicio fairSelect que permite que se atiendan mensajes por orden de fecha.

Es necesario también crear las guardas necesarias, esto se consigue mediante un array de booleanos. Para guardarlos utilizamos la llamada:

```
servicios.fairSelect(guardas);
```



Ejemplo

Suponemos que inicialmente tenemos 2 datos y que el consumidor quiere tres (envía una petición <cR1, 3>), no podemos certificar la CPRE en el canal luego, en principio, habría una opción: poner el multibuffer de extracción a cierto. En este caso no podríamos hacer Select con un vector de booleanos así que haríamos fairSelect sin parámetros. Una opción para solucionar el problema sería guardar las peticiones que no se

pueden devolver en una estructura de datos. Así guardaríamos la petición de cR1 y otra de cRm.

En este momento P1 decide enviar dos datos y Pn tres datos. Podríamos decidir que se envíe el trabajo del productor, sin embargo esto provocaría un desbordamiento (el buffer de datos es de 4, por ejemplo) por tanto no cabría una petición de tres si ya hay dos en el buffer. Una solución es que el productor envíe solo el número de datos que quiere enviar. Si no caben los datos no los envía y la petición se guarda en una estructura de datos. Para poder enviar los datos se abre un canal nuevo donde enviarán los datos, este canal se bloqueará ante las peticiones que no entren. Al tener datos (y peticiones pendientes) se desbloquea un consumidor (<cR1, 3>) y, después, se permite a Pn que envíe sus datos. Un problema que surge entonces es que las peticiones de los productores se bloqueen y no se siga el orden correcto al enviar los datos (pudiendo hacer que se cuele un envío de datos que no debería). Para solucionar esto el productor enviará una tupla(<cDi, n>) donde cDi es el nombre del canal al que quiere enviar los datos y n el número de datos. Entonces el servidor tendrá un canal por cada productor que será por donde se envíen los datos y que se abrirá cuando tenga datos.